Novel Techniques for Automorphism Group Computation *

José Luis López-Presa[1], Luis Núñez Chiroque[2], and Antonio Fernández Anta[2]

[1] DIATEL-UPM, Madrid, Spain
jllopez@diatel.upm.es
[2] Institute IMDEA Networks, Madrid, Spain
{luisfelipe.nunez, antonio.fernandez}@imdea.org

Abstract. Graph automorphism (GA) is a classical problem, in which the objective is to compute the automorphism group of an input graph. In this work we propose four novel techniques to speed up algorithms that solve the GA problem by exploring a search tree. They increase the performance of the algorithm by allowing to reduce the depth of the search tree, and by effectively pruning it.
We formally prove that a GA algorithm that uses these techniques correctly computes the automorphism group of the input graph. We also describe how the techniques have been incorporated into the GA algorithm conauto, as *conauto-2.03*, with at most an additive polynomial increase in its asymptotic time complexity.
We have experimentally evaluated the impact of each of the above techniques with several graph families. We have observed that each of the techniques by itself significantly reduces the number of processed nodes of the search tree in some subset of graphs, which justifies the use of each of them. Then, when they are applied together, their effect is combined, leading to reductions in the number of processed nodes in most graphs. This is also reflected in a reduction of the running time, which is substantial in some graph families.

1 Introduction

Graph automorphism (GA), graph isomorphism (GI), and finding a canonical labeling (CL) are closely-related classical graph problems that have applications in many fields, ranging from mathematical chemistry [4,20] to computer vision [1]. Their general time-complexity is still an open problem, although there are several cases for which they are known to be solvable in polynomial time. Hence, the construction of tools that are able to solve these problems efficiently for a large variety of problem instances has significant interest. This work focuses on the GA problem, whose objective is to compute the automorphism group of an

* Research was supported in part by the Comunidad de Madrid grant S2009TIC-1692, Spanish MINECO/MICINN grant TEC2011-29688-C02-01, Factory Holding Company 25, S.L., grant SOCAM, and National Natural Science Foundation of China grant 61020106002.

input graph (e.g., by obtaining a set of generators, the orbits and the size of this group). In this paper, novel techniques to speed up algorithms that solve the GA problem are proposed. Additionally, most of these techniques can be applied to increase the performance of algorithms for solving the other two problems as well.

1.1 Related Work

There are several practical algorithms that solve the GA problem. Most of them can also be used for CL (and consequently, for GI testing). For the last three decades, *nauty* [13,14] has been the most widely used tool to tackle all these problems. Other interesting algorithms that solve GA and CL are *bliss* [6,5], *Traces* [17], and *nishe* [19,18]. Recently, McKay and Piperno have jointly released a new version of both nauty and Traces [15] with significant improvements over their previous versions. Another tool, named *saucy* [3,7,8], which solves GA (but not CL), has the advantage of being the most scalable for many graph families, since it is specially designed to efficiently process big and sparse graphs. Recently, it was shown that the combined use of saucy and bliss improves the running times of bliss for the canonical labeling of graphs from a variety of families [9].

All these tools are based on the same principles, using variants of the Weisfeiler-Lehman individualization-refinement procedure [21]. They explore a search tree, whose nodes are identified by equitable vertex partitions, using a backtracking algorithm to compute the automorphism group of the graph and, optionally, a canonical labeling. The efficiency of an algorithm depends on the speed at which it performs basic operations, like refinement, and, mainly, on the size of the search tree generated (the number of nodes of the search tree which are explored). There are two main ways to reduce the search space: pruning, and choosing a good target cell (and vertex) for individualization.

Miyazaki showed in [16] that it is possible to make nauty choose bad target cells for individualization, so its search space becames exponential in size when computing the automorphism group for a family of colored graphs. This suggests that a rigid criterion cell selector may be easily misled so that many nodes are explored, while choosing the right cells could dramatically reduce the search space. Thus, different colorings of a graph, or just differently labeled instances, may generate radically different search trees. Algorithms for CL use different criteria to choose the target cell for individualization, but these criteria must be isomorphism invariant to ensure that the search tree for isomorphic graphs are isomorphic, what is not necessary for GA. Examples of cell selectors are: the first cell, the maximum nonuniformly joined cell, the cell with more adjacencies to non-singleton cells, etc. A cell selector immune to this dependency on the coloring or the labeling would be desirable.

Pruning the search space may be accomplished using several techniques. Orbit pruning and coset pruning are extensively used by GA and CL algorithms. Perhaps, the most sophisticated pruning based on orbit stabilizer algorithms is that of the latest versions of nauty and Traces [15], that use the random Schreier method. However, when the number of generators grow, the overhead imposed

is not negligible. *Conflict propagation* is used by bliss [5] to prune brother nodes when one of them generates a conflict which was not found in the corresponding node of the first path. Conflicts may be detected at the nodes of the search tree, or during the refinement process as done by conauto [12] (for GI) and saucy [8].

Limited early automorphism detection, when a node has exactly the same non-singleton cells (in the same position) as the corresponding (and compatible) node in the first path, is present in all versions of conauto [10]. Recently, this feature has been added to saucy [8] under the name of *matching OPP pruning*. A more ambitious *component detection* was added to bliss [5] for early automorphism detection. However, components are not always easy to discover and keep track of.

1.2 Contributions

In this paper we propose a novel combination of four techniques to speed up GA algorithms, but which can be used in GI and CL algorithms as well. (Such extensions are out of the scope of this work.) These techniques can be used in GA algorithms that follow the individualization-refinement approach. One key concept that we define, and that is used by some of the proposed techniques, is the property of a partition being a *subpartition* of another partition (see the definition in Section 3).

We propose a novel approach to *early automorphism detection* (EAD) without the need of explicitly identifying components, unlike the component recursion of bliss. EAD is based on the concept of subpartition, and its correctness is proved by Theorem 2. This technique is useful, for example, when the graph is built from regularly connected sets of isomorphic components, and components which have automorphisms themselves.

A second technique which, to our knowledge, has never be used in any other GA algorithm is *backjumping* (BJ) in the search tree, under the condition that the partition of the current node is a subpartition of its parent node. In this case, if the current node has been fully explored and no automorphism has been found, instead of backtracking to its parent node, it is possible to backtrack directly to another ancestor. Specifically, to the nearest ancestor of which the current node is not a subpartition. The correctness of BJ is proved by Theorem 3. This technique helps, for example, when there are isomorphic and non-isomorphic components in a graph.

As previously stated, the target cell selector for individualization is key to yield a good search tree. We propose a *dynamic cell selector* (DCS) that tries to generate a tree in which nodes are subpartitions of their parent nodes, so the previous techniques can be applied. If that is not possible, it chooses the vertex to individualize to be the one, among a non isomorphism invariant subset of all the possible candidates, that generates the partition with the largest number of cells. DCS adapts to a large variety of graph families. Since it is not isomorphism invariant, it cannot be applied to CL. However, it can be used for GA, using a different one for CL, once the automorphism group has been computed, in a way similar to the combined use of saucy and bliss for CL proposed in [9].

The last technique proposed is *conflict detection and recording* (CDR), an improvement of the conflict propagation of bliss. Besides recording a hash for each different conflict found exploring branches of the nodes of the first path, the number of times each conflict appeared is counted. Then, if the number of times a certain conflict has been found on a node outside the first path exceeds the number of times it was found in the corresponding node of the first path, then no other branches need to be explored in this node. This technique helps in a large variety of graph families.

We have implemented the four techniques described, and integrated them into our program conauto-2.0,[3] resulting in the new version conauto-2.03. It is worth to mention that all versions of conauto process both directed and undirected graphs (in fact they consider all graphs as directed).

We have performed an analysis of the time complexity of conauto-2.03. It is easy to adapt prior analyses [12] to show that conauto-2.0 has asymptotic time complexity $O(n^3)$ with high probability when processing a random graph $G(n,p)$, for $p \in [\omega(\ln^4 n/n \ln\ln n), 1-\omega(\ln^4 n/n \ln\ln n)]$ [2]. We then show that, in the worst case, the techniques proposed here increase the asymptotic time complexity of conauto-2.03 by an additive polynomial term with respect to that of conauto-2.0. In particular, DCS can increase the asymptotic time complexity in up to $O(n^5)$, while EAD and BJ in up to $O(n^3)$. Finally, CDR does not increase the asymptotic time complexity. Hence, if conauto-2.0 has polynomial execution time, the execution time of conauto-2.03 does not become superpolynomial. Furthermore, as will be observed experimentally, in some cases the techniques added can drastically reduce the computing time.

We have experimentally evaluated the impact of each of the above techniques for the processing of several graph families, and different graph sizes for each family. To do so, we have compared the number of nodes traversed by conauto-2.0 and the number of nodes traversed when each of the above techniques is applied. Then we have compared the number of nodes traversed, and the running times of conauto-2.0 and conauto-2.03. The improvements are significant as the size of the search tree increases, and the overhead introduced is only noticeable for very small search trees.

1.3 Structure

The next section defines the basic concepts and notation used in the analytical part of the paper. In Section 3 we define the concept of subpartition and state the main theoretical properties, which imply the correctness of EAD and BJ. Then, in Section 4 we describe how these results have been implemented in conauto-2.03 and in Section 5 we evaluate the time complexity of conauto-2.03. Finally, in Section 6 we present the experimental evaluation of conauto-2.03, concluding the paper with Section 7.

[3] The original algorithm conauto [12] solves the GI problem but not the GA problem; conauto-2.0 is a modified version that computes automorphism groups and uses limited, though quite effective, coset and orbit pruning.

2 Basic Definitions and Notation

Most of the concepts and notation introduced in this section are of common use. For simplicity of presentation, graphs are considered undirected. However, all the results obtained can be almost directly extended to directed graphs.

2.1 Basic Definitions

A *graph* G is a pair (V, E) where V is a finite set, and E is a binary relation over V. The elements of V are the *vertices* of the graph, and the elements of E are its *edges*. The set of graphs with vertex set V is denoted by $\mathcal{G}(V)$. Let $W \subseteq V$, the subgraph induced by W in G is denoted by G_W. Let $W \subseteq V$ and $v \in V$, we denote by $\delta(G, W, v)$ the number of neighbors of vertex v which belong to W. More formally, $\delta(G, W, v) = |\{(v, w) \in E : w \in W\}|$. If $W = V$, then it denotes the *degree* of the vertex.

Two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are *isomorphic* if and only if there is a bijection $\gamma : V_G \to V_H$, such that $(v, w) \in E_G \iff (\gamma(v), \gamma(w)) \in E_H$. This bijection γ is an isomorphism of G onto H. An *automorphism* of a graph G is an isomorphism of G onto itself. The *automorphism group* $\mathrm{Aut}(G)$ is the set of all automorphisms of G with respect to the composition operation.

An *ordered partition* (or *partition* for short) of V is a list $\pi = (W_1, ..., W_m)$ of nonempty pairwise disjoint subsets of V whose union is V. The sets W_i are the *cells* of the ordered partition. For each vertex $v \in V$, $\pi(v)$ denotes the index of the cell of π that contains v (i.e., if $v \in W_i$, then $\pi(v) = i$). The number of cells of π is denoted by $|\pi|$. Let $A \subseteq V$, π^A denotes the partition of A obtained by restricting π to A. The set of all partitions of V is denoted by $\Pi(V)$. A partition is *discrete* if all its cells are singletons, and *unit* if it has only one cell. Let $\pi, \rho \in \Pi(V)$, then ρ is *finer* than π, if π can be obtained from ρ by replacing, one or more times, two or more consecutive cells by their union. Let $\pi = (W_1, ..., W_m)$ and $v \in W_i$, the partition obtained by *individualizing* vertex v is $\pi{\downarrow}v = (W_1, ..., W_{i-1}, \{v\}, W_i \setminus \{v\}, W_{i+1}, ..., W_m)$.

A *colored graph* is an ordered pair $(G, \pi) \in \mathcal{G}(V) \times \Pi(V)$. Partition π assigns color $\pi(v)$ to each vertex $v \in V$. Let $\pi = (W_1, ..., W_m)$, for each vertex $v \in V$, its *color-degree vector* is defined as $d(G, \pi, v) = (\delta(G, W_i, v) : i = 1, ..., m)$. A colored graph (G, π) is *equitable* if for all $v, w \in V$, $\pi(v) = \pi(w)$ implies $d(G, \pi, v) = d(G, \pi, w)$. (I.e., if all vertices of the same color have the same number of adjacent vertices of each color.) The notion of isomorphism and automorphism can be extended to colored graphs as follows. Two colored graphs (G, π) and (H, ρ) are isomorphic if there is an isomorphism γ of G onto H, such that $\gamma(v) = w$ implies $\pi(v) = \rho(w)$.

Two equitable colored graphs $(G, \pi) \in \mathcal{G}(V_G) \times \Pi(V_G)$ and $(H, \rho) \in \mathcal{G}(V_H) \times \Pi(V_H)$ are *compatible* if and only if (1) $|\pi| = |\rho| = m$; (2) let $\pi = (W_1, ..., W_m)$ and $\rho = (W'_1, ..., W'_m)$, then for all $i \in [1, m]$, $|W_i| = |W'_i|$; (3) and for all $v \in V_G$, $w \in V_H$, $\pi(v) = \rho(w)$ implies $d(G, \pi, v) = d(H, \rho, w)$. Note that, if two colored graphs are not compatible, then they can not be isomorphic.

2.2 Individualization-Refinement and Search Trees

Most algorithms for computing GA or CL use variants of the Weisfeiler-Lehman individualization-refinement procedure [21]. This procedure requires two functions: a *cell selector* and a *partition refiner*. A *cell selector* is a function S that, given a colored graph (G, π), returns the index i of a cell $W_i \in \pi$ such that $|W_i| > 1$. A *partition refiner* is an isomorphism-invariant function R that, given a colored graph (G, π), returns (G, π) if it is already equitable. Otherwise, it returns an equitable colored graph (G, ρ) such that ρ is finer than π.

The automorphism group of a graph is usually computed by traversing a search tree in a *depth-first* manner. A *search tree* of a graph $G \in \mathcal{G}(V)$ is a rooted tree $\mathcal{T}(G)$ of colored graphs defined as follows.

1. The root of $\mathcal{T}(G)$ is the colored graph $R(G, (V))$[4].
2. Let (G, π) be a node of $\mathcal{T}(G)$. If π is discrete, it is a leaf node.
3. Otherwise, let $\pi = \{W_1, ..., W_m\}$ and assume $S(G, \pi) = j, j \in [1, m]$, and $W_j = \{v_1, ..., v_k\}$ (recall that $|W_j| > 1$ from the definition of a cell selector). Then, (G, π) has exactly k children, where the ith child is $(G, \pi_i) = R(G, \pi{\downarrow}v_i)$.

A *path* in $\mathcal{T}(G)$ starts at some internal (non-leaf) node and moves toward a leaf. A path can be denoted as $\pi_0[v_1\rangle\pi_1...[v_k\rangle\pi_k$, indicating that, starting at node (G, π_0) and individualizing vertices $v_1, ..., v_k$, node (G, π_k) is reached. The *depth* (or *level*) of a node in $\mathcal{T}(G)$ is determined by the number of vertices which have been individualized in its path from the root. Thus, if (G, π_0) is the root node, then π_0 is the partition at level 0, and π_k is the partition at level k. The first path traversed in $\mathcal{T}(G)$ is called the *first-path*, and the leaf node of the first-path is called the *first-leaf*.

Theorem 1. *Let $G = (V, E)$ be a graph. Let (G, π) and (G, ρ) be two compatible leaf-nodes in $\mathcal{T}(G)$. Then, mapping $\gamma : V \to V$ such that, for all $v \in V$, $\pi(v) = \rho(\gamma(v))$ is an automorphism of G.*

Proof. Direct from the definition of compatibility among colored graphs, and the fact that, since (G, π) and (G, ρ) are leaf-nodes, all their cells are singleton.

3 Correctness of EAD and BJ

In this section we define specific concepts needed to develop our main results, like the concept of the *kernel* of a partition, and that of a partition being a *subpartition* of another partition. Then, we state theorems that prove the correctness of the EAD and BJ techniques.

We start by defining the *kernel* of a partition, which intuitively is the subset of vertices in non-singleton cells with edges to other vertices in non-singleton cells, but not to all of them. More formally, we can define the kernel as follows.

[4] We write $R(G, (V))$ and $S(G, \pi)$ instead of $R((G, (V)))$ and $S((G, \pi))$ to avoid duplicated parentheses.

Definition 1. *Let* $(G, \pi) \in \mathcal{G}(V) \times \Pi(V)$ *be an equitable colored graph,* $\pi = (W_1, ..., W_m)$ *and* $W = \bigcup_{i:|W_i|>1} W_i$*. Then, the* kernel *of partition* π *is defined as* $\kappa(\pi) = \{v \in W : \delta(G, W \setminus \{v\}, v) \in [1, |W| - 1]\}$*. The* kernel complement *of* π *is defined as* $\overline{\kappa}(\pi) = (V \setminus \kappa(\pi))$*.*

Now we can define the concept of a subpartition of another partition.

Definition 2. *Let* (G, π) *and* (G, ρ) *be two equitable colored graphs such that* ρ *is finer than* π*. Then,* ρ *is a* subpartition *of* π *if and only if each cell in the kernel of* ρ *is contained in a different cell of* π*. (I.e.,* $\rho^{\kappa(\rho)} = \pi^{\kappa(\rho)}$*.)*

The next result allows for *early automorphism detection* (EAD) when, at some node in the search tree, the node's partition is a subpartition of an ancestor's partition. In practice, it limits the maximum depth in the search tree, necessary to determine if a path is automorphic to a previously explored one.

Definition 3. *Let* $G \in \mathcal{G}(V)$ *and* $\mathcal{T}(G)$ *its search tree. Let* (G, π_k) *be a node of* $\mathcal{T}(G)$*. Let* (G, π_l) *and* (G, ρ_l) *be two descendants of* (G, π_k) *such that (1) they are compatible, and (2)* π_l *and* ρ_l *are subpartitions of* π_k*. Let* $\pi_l = (W_1, ..., W_m)$ *and* $\rho_l = (W'_1, ..., W'_m)$*. For all* $i \in [1, m]$*, let* β_i *be any bijection from* W_i *to* W'_i*. Let us define the function* $\alpha : V \to V$ *as follows.*

- *For all* $v \in \overline{\kappa}(\pi_l)$*,* $\alpha(v) = \beta_{\pi_l(v)}(v)$*.*
- *For all* $v \in \kappa(\pi_l)$*,* $\alpha(v) = f(v)$*, where* $f(v) = v$ *if* $v \in \kappa(\rho_l)$*, and* $f(v) = f(\beta^{-1}(v))$ *if* $v \in \overline{\kappa}(\rho_l)$*.*

Theorem 2. *Let* $G \in \mathcal{G}(V)$ *and* $\mathcal{T}(G)$ *its search tree. Let* (G, π_k) *be a node of* $\mathcal{T}(G)$*. Let* (G, π_l) *and* (G, ρ_l) *be two descendants of* (G, π_k) *such that (1) they are compatible, and (2)* π_l *and* ρ_l *are subpartitions of* π_k*. Then,* (G, π_l) *and* (G, ρ_l) *are isomorphic, and* α *(as defined in Definition 3) is an automorphism of* G*.*

Interestingly, some of the properties used for early automorphism detection in other graph automorphism algorithms are special cases of the above theorem. For instance, the early automorphism detection used in saucy-3.0 is limited to the case in which all the non-singleton cells are the same in both partitions. This corresponds to the particular case of Theorem 2 in which $\overline{\kappa}(\pi_l) \cap \kappa(\rho_l) = \emptyset$, and all the cells in $\overline{\kappa}(\pi_l)$ are singleton.

The following theorem shows the correctness of *backjumping* (BJ) when searching for automorphisms. This allows to backtrack various levels in the search tree at once.

Theorem 3. *Let* (G, π_k) *be a node of* $\mathcal{T}(G)$*. Let* (G, π_l) *and* (G, ρ_l) *be two compatible descendants of* (G, π_k)*. Let* (G, π_m) *and* (G, ρ_m) *be two descendants of* (G, π_l) *and* (G, ρ_l) *respectively, such that* π_m *is a subpartition of* π_l *and* ρ_m *is a subpartition of* ρ_l*. If* (G, π_m) *and* (G, ρ_m) *are compatible but not isomorphic, then* (G, π_l) *and* (G, ρ_l) *are not isomorphic either.*

A direct practical consequence of Theorem 3 is that, when exploring alternative paths at level k, if a level m is reached that satisfies the conditions of the theorem, it is not necessary to explore alternative paths at level l. Instead, it is possible to backjump directly to the closest level $j \in [k, l)$ such that ρ_m is not a subpartition of ρ_j.

4 Implementation of the Techniques in *conauto-2.03*

The starting point is algorithm conauto-2.0, which is the first version of conauto that solves GA. It obtains a set of generators, and computes the orbits and the size of the automorphism group using the individualization-refinement approach. Its cell selector chooses a non-singleton cell with the largest number of adjacencies to non-singleton cells, and the one with the smallest size among them. The basic algorithm works in the following way. It starts by generating the first path, recording the positions of the individualized cells at each node of the path, for future use. Then, starting from the leaf parent, it explores each alternative branch. When a leaf node compatible with the leaf of the first path is reached, an automorphism is found and stored. Then, the algorithm moves to the parent node and explores the new branches of its subtree, which will generate paths of length two. This process continues until the root node of the search tree has been explored, using limited coset and orbit pruning.

EAD is implemented as follows. The first path is explored to find, for each non-leaf node (G, π), its nearest successor (G, ρ) which is a subpartition of (G, π). Note that a leaf node is a subpartition of all its ancestors. (G, ρ) is recorded as the search limit for (G, π). Then, when searching for automorphisms from (G, π), if a new node compatible with (G, ρ) is found, an automorphism α is inferred applying Definition 3. This requires a subpartition test which is linear in the number of cells, that will be executed, for each non-leaf node in the first path, at most as many times as the length of the path from that node to the leaf. Every time the search limit is not a leaf, a subtree is pruned.

BJ requires the execution of the subpartition test for the ancestors of each node (G, π) of the first path, until a node of which it is not a partition is found. That will be the backjump point for node (G, π). The point is recorded, and BJ can be subsequently applied with zero overhead.

EAD and BJ can only be applicable if there are nodes in the first path that satisfy the subpartition condition. Without a cell selector that favours subpartitions, they cannot be expected to be useful in general. Hence, a cell selector like DCS is needed. DCS works in the following way. At node (G, π), it first selects, as candidates, one cell in $\kappa(\pi)$ of each size and number of adjacencies to its kernel. From each such cell, it takes the first vertex v, and computes the corresponding refinement $R(G, \pi{\downarrow}v)$. If it gets a partition which is a subpartition of π, it selects that cell (and vertex) for individualization. If no such cell is found, it selects the cell (and vertex) which produces the partition with the largest number of cells. Observe that this function is not isomorphism-invariant (not all the vertices of a cell will always produce compatible colored graphs), and it has a significant cost in both time and number of additional nodes explored. However, it pays off because the final search tree is drastically reduced for a great variety of graphs, and other techniques compensate the overhead introduced.

Conflict detection and recording (CDR) requires a function to compute the hash of each conflict found, and storing a couple of integers for the hash and the counters. The cost incurred is very limited and there is a large variety of graphs that benefit from this technique.

5 Complexity Analysis

It was shown in [12] that conauto-1.0 is able to solve the GI problem in polynomial time with high probability if at least one of the two input graphs is a random graph $G(n,p)$ for $p \in [\omega(\ln^4 n/n \ln\ln n), 1-\omega(\ln^4 n/n \ln\ln n)]$. Using a similar analysis, it is not hard to show a similar result for the complexity of conauto-2.0 solving the GA problem. I.e., conauto-2.0 solves the GA problem in polynomial time with high probability if the input graph is a random graph $G(n,p)$ for $p \in [\omega(\ln^4 n/n \ln\ln n), 1-\omega(\ln^4 n/n \ln\ln n)]$.

We argue now that the techniques proposed in this work only increase the asymptotic time complexity of conauto-2.0 by a polynomial additive term. This implies that there is no risk that, if a graph is processed in polynomial time by conauto-2.0, by using these techniques it will require superpolynomial time with conauto-2.03. Let us consider each of the techniques proposed independently.

DCS only increases the execution time during the computation of the first-path. This follows since it is only used by the cell selector to choose a cell, and the cell selector is only used to choose the first-path. (Every time the cell selector returns a cell index, this index is recorded to be used in the rest of the search tree exploration.) The cell selector is called at most a linear number of times in n, where n is the number of vertices of the graph. Then, DCS is applied a linear number of times. Each time it is applied it may require to explore a linear number of branches. Each branch is explored with a call to the partition refiner function, whose time complexity if $O(n^3)$. Therefore, DCS increases the asymptotic time complexity of the execution by an additive term of $O(n^5)$.

Regarding EAD, like DCS, it requires additional processing while the first-path is created. In particular, for each partition π in the first-path, the closest partition down the path which is a subpartition of π is determined. This process always finishes, since the leaf of the first-path is a trivial subpartition of all the other partitions in the first-path. There is at most a linear number of partitions π and, hence, at most a linear number of candidate subpartitions. Moreover, checking if a partition is a subpartition of another takes at most linear time. Hence, EAD adds a term $O(n^3)$ to the time complexity of processing the first-path. On the other hand, when the rest of the search tree is explored, checking the condition to apply EAD has constant time complexity. If EAD can be applied, an automorphism is generated in linear time. Observe that if EAD were not used, then an equivalent automorphism would have been found, but at the cost of exploring a larger portion of the search tree (which takes at least linear time and may have up to exponential time complexity). Hence the application of EAD does not increase the asymptotic time complexity of exploring the rest of the search tree, and may in fact significantly reduce it.

The time complexity added by BJ to the processing of the first-path is similar to that of EAD, i.e., $O(n^3)$, since for each partition in π the task is to find the closest partition up the first-path which is not a subpartition of π (if such a partition exists). The application of BJ in the exploration of the rest of the search tree takes constant time to check and to apply, while the time complexity reduction can be exponential.

CDR on its hand involves no processing during the generation of the first-path. Then, during the exploration of the rest of the search tree, every time a conflict is detected, the hash of that conflict is computed and the corresponding counter has to be updated (see Section 4). This takes in total at most linear time. Observe that conflict detection, which takes at least linear time, has to be done in any case. Hence, CDR does not increase the asymptotic time complexity of the algorithm.

6 Evaluation of the Techniques in *conauto-2.03*

In this section, we evaluate the improvement in performance of conauto-2.0 by adding the proposed techniques. The experiments have been carried out in an Intel(R) Core(TM) i5 750 @2.67GHz, with 16GiB of RAM under Ubuntu Server 9.10. All the programs have been compiled with gcc 4.4.1 and optimization flag '-O2', and all the results have been verified to be correct. First, we evaluate the impact of each of the techniques proposed separately on the number of nodes that are explored during the search. Then, we evaluate the impact of their joint use in conauto-2.03 with respect to conauto-2.0. Finally, we compare the running times of conauto-2.03 vs. conauto-2.0. For the experiments, we have used all the graphs in our benchmark [11], which include a variety of graph families with different characteristics. It includes strongly regular graphs, random graphs, projective planes, Hadamard matrices, multiple variations of Miyazaki's

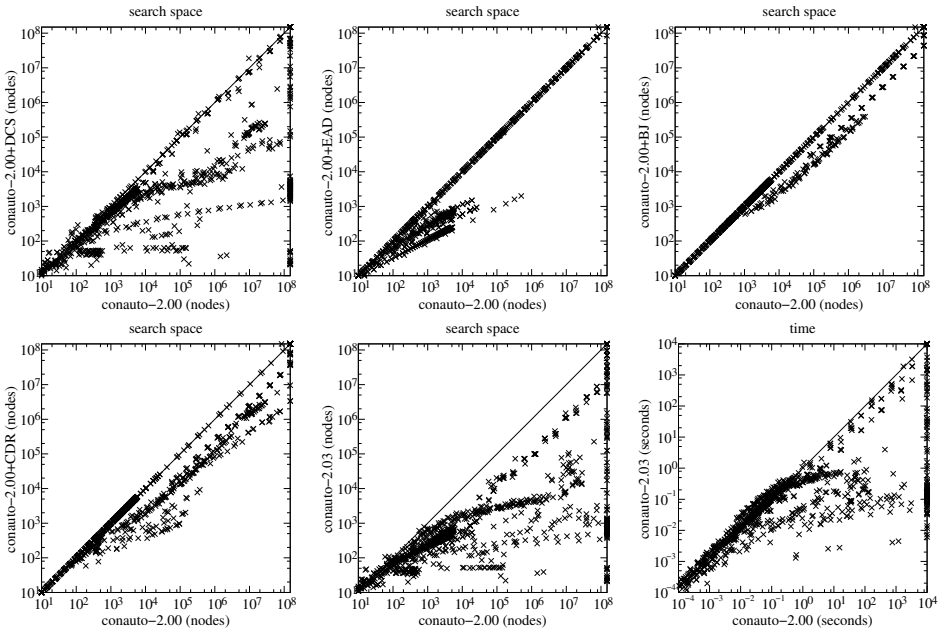

Fig. 1. Performance evaluation for the different techniques in conauto-2.0.

construction, different kinds of union graphs, etc. When counting the number of nodes of the search tree explored, each execution was stoped when the count reached 10^8. For the time comparison, a timeout of 5,000 seconds was established. When an execution reached the limit, its corresponding point is placed on boundary of the plotting area. The plots are shown in Figure 1.

As can be observed in the plots, EAD, BJ, and CDR never increase the number of nodes explored. This number slightly increases with DCS in some graphs, but only in a few executions with small search tress, and the benefit attained for most graphs is very noticeable. In fact, many executions that reached the count limit without DCS, lay within the limit when DCS is used (see the rightmost boundary of the plot). In the case of component-based graphs with subsets of isomorphic components, EAD is able to prune many branches, but with other graph families it has no visible effect. That is why the diagonal of the plot is crowded. BJ has a similar effect, but for different classes of graphs. It is mostly useful for component-based graphs which have few automorphisms, so they are complementary. EAD exploits the existence of automorphisms, and BJ exploits the inexistence of automorphisms. CDR is useful with a variety of graphs. It is mostly useful when the target cells used for individualization are big and there are few automorphisms. When DCS and/or BJ are combined with DCS, their effect increases, since DCS favours the subpartition condition, generating more nodes at which EAD and BJ are applicable. When all the techniques proposed are used (in conauto-2.03), the gain is general (big search trees have disapeared from the diagonal), and the overhead generated by DCS is compensated by the other techniques in almost all cases.

The techniques presented help pruning the search tree, but they have a computational cost. Hence, we have compared the time required by conauto-2.0 and conauto-2.03, to evaluate the computation time paid for the pruning attained. The results obtained show that the improvement in time is general and only a few runs are slower (with running time below one second). Additionally, many executions that timed out in conauto-2.0 are able to complete in conauto-2.03 (see the rightmost boundary of the *time* plot). Finally, we want to mention that exetensive experiments, not presented here for lack of space, show that only DCS increases the running time of the algorithm, and only for a few cases, while all the other techniques never increase the running times.

7 Conclusions

We have presented four techniques than can be used to improve the performance of any GA algorithm that follows the individualization-refinement approach. In particular, a new way to achieve *early automorphism detection* has been proposed which is simpler and more general than previous approaches, and its correction has been proved. These techniques have been integrated in the algorithm conauto with only a polynomial additive increase in asymptotic time complexity. We have experimentally shown that, both isolated and combined, the proposed techniques drastically prune the search tree for a large collection of graph instances.

References

1. Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Graph matching applications in pattern recognition and image processing. In *ICIP*, volume 2, pages 21–24, Barcelona, Spain, September 2003.
2. Tomek Czajka and Gopal Pandurangan. Improved random graph isomorphism. *Journal of Discrete Algorithms*, 6(1):85–92, 2008.
3. P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for cnf. In *DAC*, pages 530–534, 2004.
4. Jean-Loup Faulon. Isomorphism, automorphism partitioning, and canonical labeling can be solved in polynomial–time for molecular graphs. *Journal of Chemical Information and Computer Science*, 38:432–444, 1998.
5. Tommi Junttila and Petteri Kaski. Conflict propagation and component recursion for canonical labeling. In *TAPAS*, volume 6595 of *LNCS*, pages 151–162, 2011. 10.1007/978-3-642-19754-3_16.
6. Tommi A. Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *ALENEX*, 2007.
7. Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Symmetry and satisfiability: An update. In *SAT*, volume 6175 of *LNCS*, pages 113–127, 2010.
8. Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Conflict anticipation in the search for graph automorphisms. In *LPAR*, volume 7180 of *LNCS*, pages 243–257, 2012.
9. Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Graph symmetry detection and canonical labeling: Differences and synergies. In *Turing-100*, volume 10 of *EPiC Series*, pages 181–195, 2012.
10. José Luis López Presa. *Efficient Algorithms for Graph Isomorphism Testing.* PhD thesis, ETSIT, Universidad Rey Juan Carlos, Madrid, Spain, March 2009.
11. José Luis López-Presa, Luis Núñez Chiroque, and Antonio Fernández Anta. Benchmark graphs for evaluating graph isomorphism algorithms. Conauto website by Google sites, 2011. http://sites.google.com/site/giconauto/home/benchmarks.
12. José Luis López-Presa and Antonio Fernández Anta. Fast algorithm for graph isomorphism testing. In *SEA*, volume 5526 of *LNCS*, pages 221–232, 2009.
13. Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
14. Brendan D. McKay. The nauty page. Computer Science Department, Australian National University, 2010. http://cs.anu.edu.au/∼bdm/nauty/.
15. Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, ii. 2013.
16. Takunari Miyazaki. The complexity of McKay's canonical labeling algorithm. In *Groups and Computation II*, pages 239–256. American Mathematical Society, 1997.
17. Adolfo Piperno. Search space contraction in canonical labeling of graphs (preliminary version). *CoRR*, abs/0804.4881, 2008.
18. G. Tener. *Attacks on difficult instances of graph isomorphism: sequential and parallel algorithms.* Phd thesis, University of Central Florida, 2009.
19. G. Tener and N. Deo. Attacks on hard instances of graph isomorphism. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 64:203–226, 2008.
20. Gottfried Tinhofer and Mikhail Klin. Algebraic combinatorics in mathematical chemistry. Methods and algorithms III. Graph invariants and stabilization methods. Technical Report TUM-M9902, Technische Universität München, March 1999.
21. Boris Weisfeiler, editor. *On construction and identification of graphs.* Number 558 in Lecture Notes in Mathematics, 1976.